Réalisation de nos premiers widgets

version 1

Interface Homme-Machine : Unity (Version enseignant)

Voici les objectifs de ce sujet :

— continuer à manipuler l'IDE `Unity` ;

— créer un *widget* complexe ;

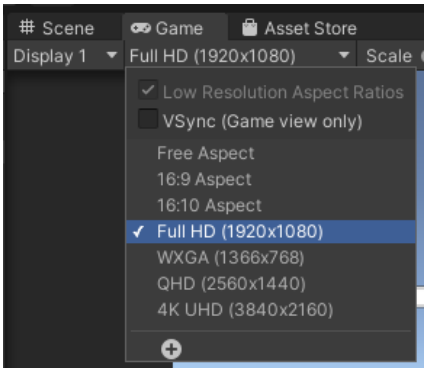
— étudier le mécanisme des préfabriqués ;

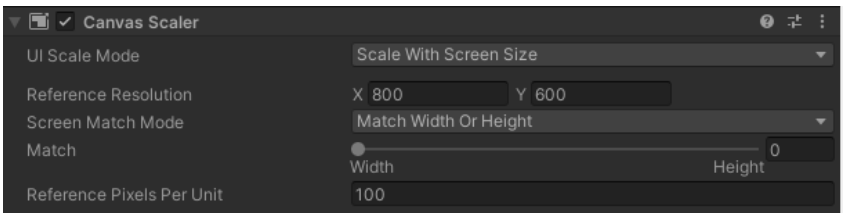
— exporter son travail : les `Unity package`.

Prélude La fois précédente, nous avons réalisé notre premier projet Unity et nous avons vu la manipulation de l'interface. La manipulation est simple (mais nécessite quelques calibrations). Nous avons réalisé entre autres notre première interface très simplifiée, reposant sur la mécanique des ancres (voire aucune pour certains d'entre vous). La mécanique des ancres n'est pas aisée, surtout lorsque les ratios de l'écran peuvent changer (réalisation d'une application `PC` et une application `Android`). Ce dernier point ne sera pas étudié dans ce TP car cela rentre dans les aspects très avancés. Cependant, pour le lecteur volontaire ou si vous finissez le TP (donc pas au début) il serait intéressant de regarder ce pointeur `https://docs.unity3d.com/2022.3/Documentation/Manual/HOWTO-UIMultiResolution.html`.

Description générale du TP

On vous demande dans un premier temps de créer un **nouveau** projet vide et de fixer une résolution raisonnable [1] de votre `Canvas`, dans le menu `Game` de la fenêtre (sinon, il s'agit de paramètres spécifiques au déploiement de votre application). Modifier également, en sélectionnant dans le `Canvas` de la hiérarchie, le composant gérant la mise à l'échelle en fonction de la résolution (`Canvas Scaler`) et respectant les paramètres (à la résolution près) de l'image de droite.





Solution

— Créer un nouveau projet de type `Universal 3D Core`.

— Dans l'onglet `Game` de l'interface, cliquer sur `Free Aspect` et sélectionner une résolution, par exemple *Full HD (1920x1080)*.

— Ajouter un `Canvas` dans la `Hierarchy`.

— Modifier le `Component Canvas Scaler` :

  — `UI Scale Mode > Scale With Screen Size`

  — `Reference Resolution` : faire correspondre à la résolution précédemment choisie

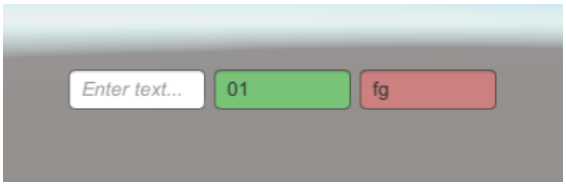
Par défaut, le `Canvas` a comme propriété *Canvas* > *Render Mode* la valeur `Screen Space - Overlay` qui signifie que le `Canvas` va couvrir tout l'écran d'affichage (par conséquent, les propriétés du `Rect Transform` sont désactivées).

Une fois cela réalisé, nous pouvons passer à la réalisation d'un widget complexe. Pour expliquer ce terme, nous allons réaliser un agglomérat de widgets existants avec un ou plusieurs scripts pour régir le comportement global du widget.

1. La résolution que j'ai dans mes versions est par défaut le `1024x768`.

1 Premier widget complexe : FormattedInputField

L'objectif de ce widget est de réaliser une zone de texte qui change de couleur selon une expression régulière particulière comme sur l'image ci-dessous, où nous avons 3 `FormattedInputField`. Ainsi, lorsque la saisie est vide (le widget de gauche), nous avons une couleur standard. La présence d'un nombre engendre une couleur (le widget au centre) et une autre couleur si l'expression régulière n'est pas présente (le widget à droite).



Solution

- Dans le `Canvas`, créer un `Panel` et le nommer `PanelInputFieldText`
  - Définir son *ancre* en `Center / Middle` pour qu'il soit placé automatiquement au centre du `Canvas`.
  - Préciser les dimensions, par exemple `Width = 1000` et `Height = 100`.
- Dans `PanelInputFieldText` :
  - Ajouter un `UI > Legacy > Input Field` et le renommer *InputFieldText1* et modifier son `Rect Transform` :
    - `Anchor Bottom Left`
    - `Pivot X = 0 ; Pivot Y = 0`
    - `Pos X = 0 ; Pos Y = 0; Pos Z = 0`
    - `Width = 300, Height = 100`
- Dupliquer ce `GameObject` et le renommer *InputFieldText2*. Le déplacer en `Rect Tranform > Pos X = 345 ; Pos Y = 0; Pos Z = 0`.
- Dupliquer ce `GameObject` et le renommer *InputFieldText3*. Le déplacer en `Rect Tranform > Pos X = 690 ; Pos Y = 0; Pos Z = 0`.

Remarque : chaque `InputFieldText` contient un `Placeholder`, qui affiche par défaut le texte : "*Enter text...*" (inutile de le modifier).

Pour cela, nous vous donnons le code suivant qui vient d'un programme `C#` basique en dehors de `Unity` :

```
Console.WriteLine("Regex␣experimentation");
string regex = "[0-9]+";
string montext = "bonjour";

if (System.Text.RegularExpressions.Regex.IsMatch(montext, regex))
       Console.WriteLine("Le␣texte␣matche␣la␣regex");
else
       Console.WriteLine("le␣texte␣ne␣matche␣pas␣la␣regex");
```

D'un point de vue conceptuel, vous devriez suivre les étapes suivantes (vous pouvez procéder autrement, mais sans garantie de bon fonctionnement) :

1. créer les attributs publics correspondant aux couleurs et les initialiser directement ;
2. créer l'attribut de la `regex` sous forme de chaîne de caractères ;
3. réaliser la `callback` lorsqu'on change le texte dans le champ de saisie ;
4. la couleur qu'il faut changer est celle du composant image.

Information

Pour réaliser le dernier point, il faut bien se rappeler de la séance précédente et de la partie sur l'`Inspector` des objets en `Unity`. En effet, nous avons différents `Components` pour le champs de saisie. Le `Component` qui nous

intéresse est l'**image** permettant de changer l'attribut `color` pour répondre à nos conditions. Pour cela, il nous suffit, à partir du `GameObject`, de récupérer l'objet souhaité :

```
Image image = gameObject.GetComponent<Image>();
```

J'insiste que `GetComponent` récupère le premier composant du type souhaité dans le `GameObject` courant, il existe une version permettant de récupérer tous les composants d'un type cible.

```
Image[] images = gameObject.GetComponents<Image>();
```

Réaliser le widget voulu, en attachant une importance à la hiérarchie dans la structure de votre projet et aux noms que vous adoptez (harmonisation et uniformisation).

Solution

Associer le script *FormattedInputFieldScript.cs* à chacun des `InputFieldTexts`.

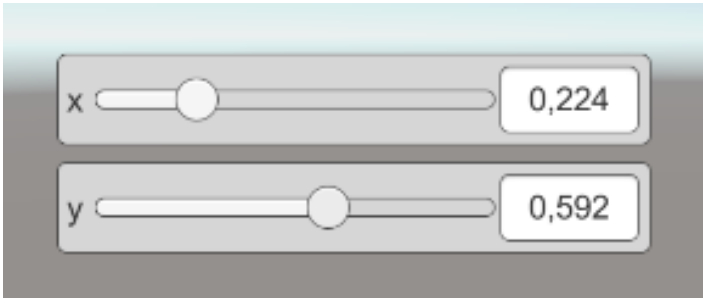
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.Text.RegularExpressions;

// Script associe aux GameObject de type "InputFieldText"

public class FormattedInputFieldScript : MonoBehaviour
{
	// Regex validant uniquement une suite de chiffres.
	// Modifiable dans l'Inspector
	public string m_Regex = "^[0-9]+$";

	// Composant InputField lie a ce GameObject
	private InputField m_InputField;

	// Si InputField vide
	private Color colorEmpty = Color.white;

	// Si texte de l'InputField ne valide pas le regex
	private Color colorError = Color.red;

	// Si texte de l'InputField valide le regex
	private Color colorValid = Color.green;


	// Start is called before the first frame update
	void Start()
	{
		m_InputField = this.GetComponent<InputField>();
		if (m_InputField == null) {
			Debug.Log("[FormattedInputFieldScript] input field = nul");
		}

		m_InputField.onValueChanged.AddListener(delegate {
			ValueChangeCheck();
		});

	}
```

```
        // Attention : cette fonction est definie independamment de la méthode Start()
        // => initialiser [m_Text] et [m_Image] dans Start() ne sert a rien
        // ==> ils sont consideres comme "null" dans cette fonction.
        // ===> on en fait des variables locales et le probleme est regle.
        private void ValueChangeCheck() {
                string m_Text = m_InputField.text;
                if (m_Text == null) {
                        Debug.Log("[FormattedInputFieldScript] text = nul");
                }
                else {
                        // Le type "string" ne supporte pas la concatenation àla Java
                        // => Debug.Log("m_Text = " + m_Text); provoque des erreurs
                        Debug.Log(string.Concat("m_Text = ", m_Text));

                        Image m_Image = this.GetComponent<Image>();
                        if (m_Image == null) {
                                Debug.Log("[FormattedInputFieldScript] image = nul");
                        }

                        if (string.IsNullOrEmpty(m_Text)) {
                                m_Image.color = this.colorEmpty;
                        }
                        else {
                                Debug.Log(string.Concat("m_Regex = ", m_Regex));

                                // Version fonctionnelle numero 1
                                /*
                                if (Regex.IsMatch(m_Text, m_Regex)) {
                                        m_Image.color = colorValid;
                                }
                                else {
                                        m_Image.color = colorError;
                                }
                                */

                                // Version fonctionnelle numero 2
// https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity5.html
// https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.match?view=net-8
                                Regex myRegExp = new Regex(m_Regex);
                                Match myMatch = myRegExp.Match(m_Text);
                                if (myMatch.Success) {
                                        m_Image.color = colorValid;
                                }
                                else {
                                        m_Image.color = colorError;
                                }
                        }
                }
        }
}
```

2 Widget : ComplexSlider

Dans la section précédente, nous avions un unique `widget`. Ici, nous allons composer un `widget` complexe à partir de 3 `widgets` basiques. Pour cela, nous allons réaliser des curseurs complexes en composant un `Text` qui servira de

`Label`, un `Slider` basique et un `InputTextField` sur les nombres pour visualiser la valeur du `Slider` et/ou la modifier manuellement. Ci-dessous, vous avez une image qui illustre deux `ComplexSliders` pour choisir des coordonnées `(x,y)`.



Information

Généralement, un widget complexe doit avoir un `Panel` à la base pour servir de fond visuel, que vous opacifierez par défaut et qui permet de hiérarchiser les 3 sous-`widgets` le composant.

Vous devez construire un tel composant puis écrire un unique script qui gouverne tous les aspects comportementaux. En particulier, lorsqu'une valeur est modifiée, cela impacte l'autre `widget` pour avoir toujours une cohérence entre le champ de saisie et le `Slider` basique.

Pour cela, nous allons exploiter la hiérarchie de notre `widget`. Il existe deux mécaniques pour retrouver les enfants :

1. *via* les fonctions de recherche basées sur leur nom ;
2. *via* les mécaniques de recherche sur un type souhaité.

Réfléchissez aux avantages et inconvénients des deux mécaniques en lisant la documentation associée à ces familles de fonctions : `https://docs.unity3d.com/ScriptReference/GameObject.html`. En particulier, que se passe-t-il si plusieurs objets sont du même type ? Ou si l'utilisateur modifie le nom d'un sous-`widget` ?

Solution

2.1 Création du Panel

- Dans le `Canvas` principal, créer un nouveau `Panel` et le nommer *PanelComplexSlider1*.
- Définir son `ancre` en `Center / Middle`.
- Préciser les dimensions, par exemple `Width = 1000 et Height = 100`.
- Déplacer *PanelComplexSlider1* en `Pos X = 0; Pos Y = -200; Pos Z = 0`.
- Dans *PanelComplexSlider1*, créer
  - un `Text avec UI > Legacy > Text` renommé *Text*
    - `Rect Transform`
      - `Anchor = bottom left`
      - `Pivot : X = 0 ; Y = 0`
      - `Pos X = Pos Y = Pos Z = 0`
      - `Width = 40 : Height = 100`
    - Text
      - `Text = x`
      - `Font Size = 36` [**ATTENTION** : si la taille est trop grance, rien n'apparaît dans le widget]
      - `Paragraph > Alignment = middle / center`
  - un `Slider` avec `UI > Slider`
    - `Rect Transform`
      - `Anchor = bottom left`
      - `Pivot : X = 0.5 ; Y = 0.5`
      - `Pos X = 450 ; Pos Y = 50 ; Pos Z = 0`
      - `Width = 700 : Height = 60`
    - `Max value` = *100*

    - Enfant `Handle Slide Area > Handle` : dans `Rect Transform`, modifier la propriété `Width` pour que la poignée du `Slider` dessine un joli cercle, égale à la hauteur du `Slider` : 60.
- un `Input Field` *InputFieldComplexSlider* avec `UI > Legacy > Input Field`
  - `Rect Transform`
    - `Anchor = bottom left`
    - `Pivot : X = 0 ; Y = 0`
    - `Pos X = 830 ; Pos Y = 20 ; Pos Z = 0`
    - `Width = 160 : Height = 60`
  - Enfant `PlaceHolder > Text` : effacer le texte par défaut *Enter text...*
  - Enfant `Text (Legacy)` : propriété `Text`
    - `Font Size = 36`. **ATTENTION** : le texte ne s'affiche pas si une taille de police est trop importante.
    - `Paragraph > Alignment = middle / center`

2.2 Script de PanelComplexSlider1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

public class ComplexSliderScript : MonoBehaviour
{
	public Slider m_Slider;
	public InputField m_InputField;

	// Start is called before the first frame update
	void Start()
	{
		m_Slider = gameObject.GetComponentInChildren<Slider>();
		m_InputField = gameObject.GetComponentInChildren<InputField>();

		Debug.Log("Slider found: " + m_Slider + " name: " + m_Slider.name);
		Debug.Log("Field found: " + m_InputField + "name: " + m_InputField.name);

		m_Slider.onValueChanged.AddListener(UpdateValueFromFloat);
		m_InputField.onEndEdit.AddListener(UpdateValueFromString);
	}

	// Update is called once per frame
	void Updte()  { }

	public void UpdateValueFromFloat(float value)
	{
		Debug.Log("float value changed: " + value);
		if (m_InputField) { m_InputField.text = value.ToString(); }
		else Debug.Log("m_InputField not found");
	}

	public void UpdateValueFromString(string value)
	{
		Debug.Log("string value changed: " + value);
		try
		{
			float ff = float.Parse(value);
			if (m_Slider && m_Slider.value != ff) { m_Slider.value = ff; }
		}
		catch(System.Exception e) {
			Debug.Log("error: " + e);
		}
	}
}
```

3 Exporter son travail

Bravo, vous avez fait le plus gros. Demander à l'enseignant qu'il vérifie votre développement ou critique vos noms et autres petits détails.

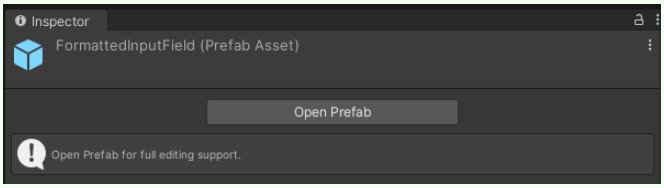
3.1 Préfabriqué

L'intéret de développer un widget est de pouvoir le réutiliser plusieurs fois sans devoir faire plusieurs manipulations identiques ou des `copier-coller`. Pour cela, `Unity` a la possibilité de créer des `Préfabs`, une sorte de sauvegarde de votre réalisation au sein d'un projet.

La procédure est assez simple : lorsque vous avez fini un `widget` qui n'a pas de dépendance extérieure, c'est-à-dire que le `widget` est autonome (sinon les dépendances risquent aussi d'être sauvegardées) : il suffit de glisser le `GameObject` de la hiérarchie vers la zone des `Assets`. Ainsi l'icône du `Gameobject` devient bleue ! C'est ainsi qu'on sait qu'il s'agit d'un `Prefab`.

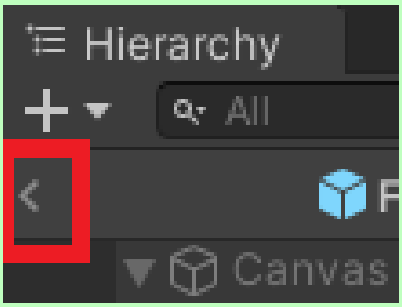
Information

Lorsque vous sélectionnez un `Prefab` dans la zone des `Assets`, vous avez un nouveau bouton dans l'`Inspecteur` qui vous permet de modifier ce `Prefab`.



Attention

Quand vous modifiez un `Prefab`, vous êtes dans un environnement particulier. Il est essentiel de revenir au plus vite dans le mode standard, en particulier pour sauvegarder et ainsi éviter des petits problèmes.
Pour quitter, il faut cliquer sur le bouton "Retour" (noté par $'<'$) au niveau de la hiérarchie.



Réaliser des `Prefabs` de vos widgets en les ayant nettoyés si besoin AVANT ! Ensuite, instancier pour expérimenter deux ou trois de vos `Prefabs`. Pour cela, il suffit de glisser votre `Prefab` depuis l'`Assert` vers la vue 3D ou la hiérarchie à l'endroit souhaité.

Solution

Un `Prefab` est un container représentant la sauvegarde d'un `Component`, que l'on peut réutiliser dans d'autres projets.
Il suffit de sélectionner le composant qui nous intéresse : ici, *PanelComplexslider1*, et de le déplacer dans le `Project`.

Attention : veiller à ce que ce composant n'ait pas de dépendance extérieure, sinon ces dernières seront également sauvegardées dans le `Prefab`.

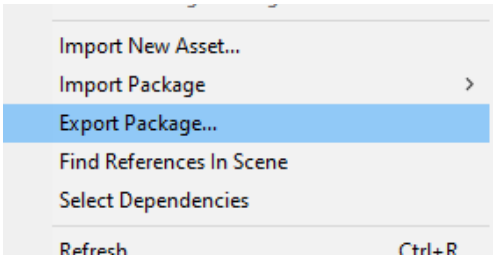
Remarque : si l'on sélectionne ce `Prefab`, l'`Inspector` affiche un bouton `Open` qui modifie l'interface principale pour lister uniquement la hiérarchie de ce `Prefab`. Pour revenir à la hiérarchie complète, on clique sur le symbole '<' à gauche du nom du `Prefab` dans la hiérarchie.

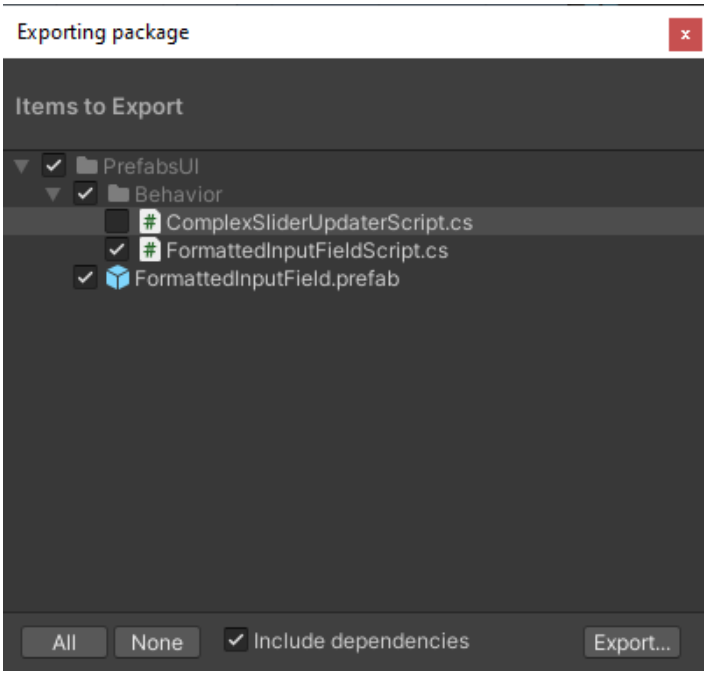
Dupliquer un Prefab et le modifier La duplication est simple : on sélectionne le `Prefab` dans le `Project` et on le déplace dans la fenêtre d'affichage ou dans la hiérarchie. Prendre soin de l'insérer comme enfant du `Canvas`. Si l'on veut modifier un attribut d'un `Prefab`, on peut sélectionner ce dernier dans le `Project`, puis faire un copier-coller. Un `Prefab` identique au précédent est créé (avec un nouveau nom) et on peut modifier ses attributs, indépendamment du `Prefab` original.

3.2 Export

Maintenant que vous avez la possibilité de sauvegarder et réutiliser votre production au sein d'un projet, il est fréquent de devoir les exporter pour les utiliser dans d'autres projets. Pour cela, il faut exporter des `Prefabs` dont vous connaissez les dépendances, en sélectionnant votre `Prefab` dans les `Assets` et en faisant un clique droit dessus (attention, on vous demande de tester l'export d'un widget dans un premier temps, puis plusieurs widgets dans un second temps afin que vous compreniez les dépendances). Dans le menu contextuel, sélectionner `'Export Package...'`.

Dans la nouvelle fenêtre, vous devez sélectionner les bonnes dépendances de votre widget. En particulier, `Unity` sélectionne par défaut tous les scripts sans distinction, car il n'arrive pas à calculer les dépendances correctement. Vous devez donc sélectionner les bonnes dépendances de scripts manuellement. Puis, cliquez sur le bouton `Export...` pour sauvegarder le package `Unity`.





Solution

- Sélectionner un `Prefab` dans l'onglet `Project`
- clique droit : sélectionner `Export...` et ne cocher que les dépendances nécessaires :
  - le `Prefab` lui-même ;
  - le script *ComplexSliderScript.cs*
- puis valider l'export en précisant son nom, par exemple *MyPrefabComplexSlider* : un fichier du nom *MyPrefabComplexSlider.unitypackage* est créé.

3.3 Import

Pour tester nos paquets `Unity` créés à l'étape précédente, c'est très simple ! Il suffit de créer un nouveau projet `Unity` et glisser votre fichier `.unitypackage` dans les `Assets`, puis se laisser guider par le menu.

Information

Une alternative consiste à faire juste un clique droit dans la zone des `Assets` et cliquer sur l'import d'un paquet personnalisé.

Votre widget est prêt à l'emploi. La mécanique des `Unity packages` est l'une des façons de se partager le travail lorsqu'on est plusieurs sur un même projet et que cela s'y prête bien.

Solution

- Créer un nouveau projet `Unity`
- Dans le menu principal, cliquer sur `Assets > Import Package > Custom Package...`
- Sélectionner *MyPrefabCmplexslider.unitypackage* (on peut sélectionner tout ou partie du package)
- Cliquer sur `Import` après la sélection
- Un `GameObject` nommé *PanelComplexSlider1* apparaît dans `Project > Assets`.
- Il n'est pas directement visible : créer un `Canvas` et intégrer *PanelComplexSlider1* comme son enfant.
- `PanelComplesSlider` est un objet 3D ! Modifier sa position pour le "coller" au `Canvas (Rect Transform > Pos Z = 0)`.
- Passer dans le `Game View` et tester le `ComplexSlider` ⇒ penser à remettre la résolution du projet original, le `Slider` risque de ne pas être adapté à la résolution du nouveau projet...

4 Toujours plus loin

4.1 Des Prefabs partout !

Reprendre votre TP précédent ou un nouveau projet qui exploite vos nouveaux widgets, histoire de construire une application complète et comprendre que votre `Prefab` doit bien être autonome pour embarquer uniquement son comportement intrinsèque !

Solution

On peut reprendre le projet sur le système solaire et vérifier que le `ComplexSlider` s'intègre correctement (l'interface commence à être "bouchée" mais bon...)

4.2 Retour sur la multi-résolution

En reprenant l'hyperlien cité en introduction, vérifier que les *ancres* des éléments du `Canvas` sont bien définies puis jouer avec le composant `Canvas Scaler` pour tester le positionnement de ces éléments.

Tester également les résolutions proposées dans le `GameView`. Enfin, créer une résolution personnalisée (imiter un mode "*Portrait*" par exemple) et vérifier si les widgets sont toujours positionnés correctement.

Solution

Documentation multiresolution.

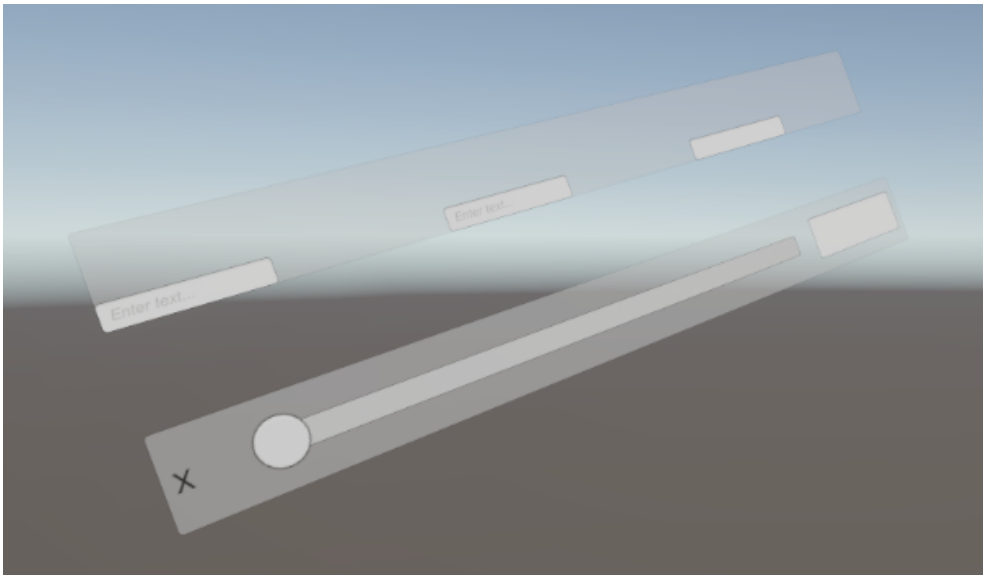
La documentation du `Canvas Scaler` permet de jouer sur plusieurs paramètres, notamment :

- `Constant Pixel Size` : tailles indépendantes de la résolution.
- `Scale With Screen Size` : à partir de la `Reference Resolution`, les dimensions des `widgets` s'adaptent en fonction de la largeur ou de la hauteur du `Canvas` (valeur entre 0 et 1).
- `Constant Physical Size` : s'appuie sur une unité physique pour recalculer les dimensions.

De plus, dans l'onglet `Resolution` qui permettait de sélectionner "*Full HD*" (entre autres), on peut créer sa propre résolution, pour imiter le mode *Paysage* : essayer par exemple *1080x1920* pour vérifier les dimensions et la disposition des `widgets`.

4.3 Orientation du Canvas

À partir du lien UICanvas, expérimenter les modes de rendu `"Screen Space - Camera"` et `"World Space"` pour modifier l'orientation et la position du `Canvas` et de ses enfants (fig. ci-dessous). Noter que la caméra associée devra peut-être devoir être mise à jour pour visualiser le `Canvas`.



Solution

On veut faire pivoter le `Canvas` pour obtenir le même effet que celui montré dans l'hyperlien.

- Sélectionner le `Canvas` dans la hiérarchie
  - Propriété `Render Mode = World Space` => `Unity` se plaint si aucune caméra n'est associée à ce mode
  - Propriété `Event Camera` : faire glisser une des caméras de la scène (par exemple *Main Camera* dans cette propriété
- Composant `Rect Transform`
  - `Pos Z` : modifier cette valeur pour éloigner le `Canvas` de la caméra ;
  - `Rotation` : modifier la valeur sur chacun des axes pour observer différents effets.

Remarques :

- On peut modifier la rotation de chaque enfant du `Canvas` indépendamment des autres.
- Si on utilise la valeur `Canvas > Render Mode = Screen Space - Camera` (toujours avec la caméra associé au `Canvas`), le positionnement du `Canvas` (*i.e.* `Component Rect Transform`) est désactivé. Mais on peut toujours modifier le positionnement de ses enfants.